METHOD AND COMPUTER PROGRAM PRODUCT FOR PROVIDING A META-DATA PROGRAMMING LANGUAGE LEVEL INTERFACE

BACKGROUND OF THE INVENTION

[0001] The present disclosure relates generally to a method for providing a meta-data programming language level interface and in particular, to a method for providing a meta-data programming language interface that may be accessed during program runtime.

[0002] Today's Internet driven economy has accelerated users' expectations for unfettered access to information resources and transparent data exchange among applications. One of the key issues limiting data interoperability today is that of incompatible meta-data. Meta-data is information about other data, or simply data about data. Meta-data is typically utilized by tools, databases, applications and other information processes to define the structure and meaning of data objects. Unfortunately, most applications are designed with proprietary schemes for modeling meta-data. Applications that define data using different semantics, structures and syntax are difficult to integrate, impeding the free flow of information access across application boundaries. This lack of meta-data interoperability hampers the development and efficient deployment of numerous business solutions (e.g., data warehousing, business intelligence, software development).

[0003] The Common Object Request Broker Architecture (CORBA) specification enforces a language neutral interface, or meta-data, definition for remote distributed object services. There are at least two ways in which a CORBA enabled object or service can be defined. First, by defining a distributed service's interface in an interface definition language

(IDL) and deriving each language level interface via an IDL compiler for each language. Second, a CORBA enabled object or service may be defined by taking an existing service defined already in a particular programming language (e.g., Java, C++, Ada) and constructing a language neutral interface definition from the programming language level interface. In the second scenario, a particular programming language level interface must be examined in a purely abstract manner without any programming language specific validation of referenced types in an interface definition so that a language neutral interface definition can be derived from it.

[0004] Currently, many object-oriented programming languages define an interface and/or abstract class definition that describes the API provided by a program module service. This is done to clearly separate the abstract interface from the implementation class that implements one or more of the abstract interface APIs. This type of programming separation technique allows developers to provide distributed objects that expose their remote interfaces in a purely abstract manner. This allows distributed clients of the distributed objects to use that abstract interface information to learn about the service. This information may be utilized for many purposes including dynamic method invocation, client-side proxy of target service generation and client-side tooling for distributed object services. But currently, in order to achieve such interface to implementation separation in a distributed environment, the developer would be required to take the existing interface definitions already defined at its language level and redefine them as distributed object definitions and work back down into the language level to piece them together.

SUMMARY OF THE INVENTION

[0005] In one embodiment, a method for providing a meta-data programming language level interface is disclosed. The method includes receiving an object name from a client program via a meta-data retrieval API, where the object name corresponds to an object located in a runtime environment that includes one or more methods. Meta-data associated with the object is requested from the runtime environment. Meta-data is received for each method included in the object. The meta-data for each method is transmitted to the client program via the meta-data retrieval API.

[0006] In another embodiment, a computer program product for a method for providing a meta-data programming language level interface is disclosed. The computer program product comprises a storage medium readable by a processing circuit and storing instructions for execution by the processing circuit for performing a method. The method includes receiving an object name from a client program via a meta-data retrieval API, where the object name corresponds to an object located in a runtime environment that includes one or more methods. Meta-data associated with the object is requested from the runtime environment. Meta-data is received for each method included in the object. The meta-data for each method is transmitted to the client program via the meta-data retrieval API.

BRIEF DESCRIPTION OF THE DRAWINGS

[0007] Referring to the exemplary drawings wherein like elements are numbered alike in the accompanying Figures:

[0008] FIG. 1 is a block diagram of high level exemplary meta-data provider classes for providing a meta-data programming language level interface in the Java programming language;

[0009] FIG. 2 is a block diagram of high level exemplary meta-data provider classes for providing a meta-data programming language level interface in the C++ programming language;

[0010] FIG. 3 is a block diagram of exemplary objects and interfaces that may be utilized in an exemplary embodiment of the present invention; and

[0011] FIG. 4 is an exemplary process flow for providing a meta-data programming language level interface.

DETAILED DESCRIPTION OF THE INVENTION

[0012] An exemplary embodiment of the present invention provides a tightly integrated programming language level interface/service meta-data application programming interface implementation within object-oriented or service-oriented programming languages. A client program may introspect the meta-data of any running object in a purely abstract manner using these API methods. An exemplary embodiment of the present invention provides an integrated way for any object-oriented programming language to provide abstract interface level meta-data within its program language runtime environment rather than relying on a separate interface definition language for a given programming language level interface definition. This support allows programming languages that provide this functionality to achieve bottom-up mapping of their programming language level interface definition to more abstract distributed object level interface definition. This bottom-up mapping technique allows any running object with this

programming language level interface/service meta-data retrieval API support to be dynamically introspected during run-time. This includes exposing the running object interface/service meta-data without prior knowledge to the running object API.

[0013] In addition, this type of programming separation technique lends itself to providing distributed objects that expose their remote interfaces in a purely abstract manner. This allows distributed clients of the distributed objects to use that abstract interface information to learn about the service that it is planning to use. The information may be utilized for many purposes including dynamic method invocation, client-side proxy of target service generation, and client-side tooling for distributed object services. In alternate exemplary embodiments of the present invention the same functions are provided within a non-object oriented programming language that clearly define its services in a distributed manner for access by distributed clients within their own programming language.

[0014] Exemplary embodiments of the present invention, by utilizing purely abstract interfaces obtained through a meta-data retrieval application API, may dynamically introspect the meta-data within a programming language specific environment and provide an abstract interface to a client environment dynamically. The client tooling program may then assemble client method invocations dynamically. In a similar context, with this type of support, a client tooling environment may also display the meta-data information of any programming language specific services in a purely abstract language neutral manner. An exemplary embodiment of the present invention describes the manner in which programming languages may furnish this purely abstract interface definition without any unnecessary language specific validation and/or runtime checks. This definition may be furnished by program language providers within a programming language through clearly defined APIs for retrieving meta-data.

[0015] FIG. 1 is a block diagram of high level exemplary meta-data provider classes for providing a meta-data programming language level interface in a Java programming language. In order for a programming language provider to furnish purely abstract interface definitions, it must first provide a meta-data retrieval API to create and return the running object instance's purely abstract interface meta-data as a meta-data class instance that may be used to retrieve other purely abstract meta-data information. Examples of other purely abstract meta-data information include method signatures, field declarations and exceptions. Each meta-data class instance that represents information such as method, field, exception, and so on, should contain only purely abstract type information.

[0016] FIG. 1 depicts a Java programming language interface source 102 that defines the Java interfaces in a human readable format. The Java interface source 102 is compiled into a Java interface meta-data class 104. The Java interface meta-data class 104 is an instantiated class instance that represents the corresponding interface source's 102 abstract interface definition during run-time. The Java interface meta-data class 104 returns the Java interface for any object instantiated by the programming language. The Java client 110 has access to the Java interface meta-data class 104, the Java method meta-data class 108, the Java field meta-data class 106, the Java interface meta-data type 112 and the Java parameter meta-data type 114 via a meta-data retrieval API. As depicted by the arrows, the Java method meta-data class 108 and the Java field meta-data class 106 may be associated with the Java interface meta-data class 104 (e.g., by inheritance). Meta-data types in addition to parameters and interfaces may be accessed via the API. Additional meta-data types may include exceptions and return types.

[0017] Given a running Java object, a Java client which has a handle to this Java object may discover its interface/service meta-data which is originally declared in its Java interface source code 102 by using an API. The Java client will use an API built into the Java object itself to initially retrieve a Java interface meta-data class 104. Then, the Java interface meta-data class 104, will request for its containing methods, one or more instances of the Java method meta-data class 108 as well as its fields, represented as one or more instances of the Java field meta-data class 106. Each of these classes (Java interface meta-data class 104, Java field meta-data class 106, Java method meta-data class 108) will hold abstract data such as: its type (in the case of the Java interface meta-data class 104 the type of its Java interface meta-data); its return type as a Java interface meta-data type 112 (in the case of the Java method meta-data class 108, the return type of its Java method meta-data which is an instance of Java interface meta-data); its Java parameter meta-data parameter type 114 (in case of 108 - parameter type of its Java method meta-data); and its field type as a Java interface meta-data type 112 type (in the case of the Java field meta-data class 106, the type of its Java field meta-data which will be an instance of Java interface meta-data or a primitive data type). Note that the Java parameter meta-data type 114 and an optional additional exception meta-data type may be implemented as sub-interfaces of the Java interface meta-data type 112.

[0018] FIG. 2 is a block diagram of high level exemplary meta-data provider classes for providing a meta-data programming language level interface in a C++ programming language. FIG. 2 includes a C++ abstract class source 202 that contains the abstract class definition, or meta-data, in a human readable format. The C++ abstract class meta-data class 204 represents an instance within a programming language, in this case C++, that returns the C++ interface for any object instantiated by the programming language. The C++ client 210 has access to the C++ abstract class meta-data class 204, the C++ method meta-data class

208, the C++ field meta-data class 206, the C++ abstract class meta-data type 212, the C++ parameter meta-data type 214 and the C++ exceptions meta-data type 216 via a meta-data retrieval API. As depicted by the arrows, the C++ method meta-data class 208 and the C++ field meta-data class 206 may be associated with the C++ abstract class meta-data class 204 (e.g., by inheritance). Meta-data types in addition to parameters, abstract classes and exceptions may be accessed via the API. An additional meta-data types may include a return type.

[0019] Given a running C++ object, a C++ client which has a handle to this C++ object may discover its abstract/service meta-data which is originally declared in its C++ abstract source code 202 by using an API. The C++ client will use an API built into the C++ object itself to initially retrieve a C++ abstract meta-data class 204. Then, the C++ abstract meta-data class 204, will request for its containing methods, one or more instances of the C++ method meta-data class 208 as well as its fields, represented as one or more instances of the C++ field meta-data class 206. Each of these classes (C++ abstract meta-data class 204, C++ field meta-data class 206, C++ method meta-data class 208) will hold abstract data such as: its type (in the case of the C++ abstract meta-data class 204 the type of its C++ abstract meta-data); its return type as a C++ abstract meta-data type 212, (in the case of the C++ method meta-data class 208, the return type of its C++ method meta-data which is an instance of C++ abstract meta-data); its C++ parameter meta-data parameter type 214 (in case of 208 - parameter type of its C++ method meta-data); and its field type as a C++ abstract meta-data type 212 (in the case of the C++ field meta-data class 206, the type of its C++ field meta-data which will be an instance of C++ abstract meta-data or a primitive data type). Note that the C++ parameter meta-data type 214 and an optional additional exception meta-data type may be implemented as sub-abstracts of the C++ abstract meta-data type 212.

[0020] FIG. 1 and FIG. 2 depict exemplary meta-data provider classes for providing a meta-data programing language level interfaces for both Java and C++ programming languages. These are meant to be examples of two programing languages that may support a meta-data retrieval API. Other languages (e.g., COBOL, Eiffel, Pascal, Ada) may also be implemented with exemplary embodiments of the present invention.

[0021] In an exemplary embodiment of the present invention, the Java programming language API may be provided by utilizing the bytecode of the language level interface class and providing an API to retrieve meta-data information for the given interface class in a purely abstract manner (e.g., via string data). Java class meta-data information such as field(s) and/or method(s) may be reflected without the required validation of any referenced classes (e.g., required when Java Reflection API is used). This allows applications that require field(s)/method(s) information at an object interface level for code generation (e.g., ejbdeploy, rmic) and/or application assembly or deploy tools (e.g., application assembly tool, application server deployment tool) to be able to retrieve such information without burdening itself by having to load all referenced classes during Java Reflection API operations.

[0022] Exemplary embodiments of the present invention provide a new static reflection API for retrieving such meta-data information on a Java class. The implementation may be written in the Java programming language, although it may be written in other programming languages and hooked into the Java programming language via an API. In an exemplary embodiment, the byte array of the Java class is reflected along with a ClassLoader instance to be utilized to search for the requested class. An exemplary embodiment of the present invention creates the internal data structure of the input class byte array (also known as the bytecode) and extracts the field_info section along with the method_info section of the

bytecode. Then it processes these bytecode fields along with the bytecode constant pool (cp_info) to find any classes that are referred to by the info sections. Once they are all found, a Java object is constructed that holds all necessary information for representing a field and/or method instance.

[0023] An exemplary embodiment of utilizing a meta-data programming language level interface in the Java programming language follows. These same concepts may be applied to other programming languages (e.g., C++, Eiffel, COBOL). FIG. 3 is a block diagram of exemplary objects and interfaces that may be utilized in an exemplary embodiment of the present invention. FIG. 3 depicts a live, running java object called the superman object 310 that has been initiated from a Java client 110. The superman object 310 inherits from the human interface 302 and from the bird interface 306. The human interface 302 includes three methods 304: walk(); talk() and write(). The bird interface 306 includes the methods 306: fly() and talk(). In an exemplary embodiment of the present invention, the client code has created an instance of the Superman class and wants to inquire about the meta-data associated with the human interface 302 and bird interface 306.

[0024] The human interface 302 may be defined in a Java programming language as a Java source file. The human interface 302 source file may include:

```
// Human interface - enforces walk, talk, write methods
interface Human {
        Destination walk (Destination target);
        Human talk (Human target);
        WriteableObject write (WriteableObject target);
}
```

Similarly, the bird interface 306 source file may include:

```
// Bird interface - enforces fly, talk methods
interface Bird {
        Destination fly (Destination target);
        Bird talk (Bird target);
}
```

In addition, the Superman class may be defined as:

```
// Superman class - which is instantiable during runtime
class Superman implements Human, Bird {
        String name = "Kent, Clark";
        public Superman (String name) {
                this.name = name;
        }
}
```

Note that the implementation of the human interface methods 302 and the bird interface methods 308 would also be defined in a Java Source file. These interface methods have been omitted for brevity.

[0025] Also, in this example, a client MAIN function instantiates the Superman class and then disguises it as an interface. Disguising a class as an interface is typical in a distributed environment where the client almost always only works with interfaces since the instantiated object on the client side is a proxy that hides the internal workings of the methods in the class. In addition to disguising the Superman class, the MAIN function calls an exemplary embodiment of a Java interface meta-data class 104 retrieval API, or hook, called "interface" on a live Java object to retrieve the Human interface and Bird interface which the Superman class implements to instantiate a "java.lang.Interface" class for both Human and Bird interface respectively. Input to the "printInterface" method includes an interface name. Output from the "printInterface" method includes meta-data (e.g., the interface definition) associated with the

interface that was input. An exemplary embodiment of a client MAIN function, written in a Java language follows:

```
// Client code being driven to use Superman object and its interfaces
public static void main (String[] args) {

        // Creating a Superman object
        Superman clark = new Superman("Kent, Clark");

        // Superman disguised as a Human
        Human man = (Human) clark;

        // Superman disguised as a Bird
        Bird bird = (Bird) clark;

        // java.lang.Interface class that represents the Human (meta-data) interface
        printInterface(man.interface);

        // java.lang.Interface class that represents the Bird (meta-data) interface
        printInterface(bird.interface);
}
```

[0026] The printInterface interface method that is invoked prints out interface information to a screen. This method may be utilized to duplicate the interface definition dynamically during runtime. Other interface methods may be created from the java.lang.Interface class and utilized to duplicate interface definitions, or interface meta-data during runtime. An exemplary embodiment of a printInterface method follows:

```
private static void printInterface(java.lang.Interface intf)

        //Introspect input java.lan.Interface using a meta-data programming
        // language level interface
        Method[] intfMethods = intf.getMethods();

        System.out.println("interface " + intf.getName() + " {");

        // Print out interface dynamically during runtime
        for (inst i=0; i,intfMethods.length; i++) {
                String methodName = intfMethods[].getName();
                String returnType = intfMethods[i].getReturnType();
                String[] paramterTypes = intfMethods[i].getParameterTypes();

                System.out.print("\t" + returnType + " " + methodName + "(");
                for (intf j = 0; j<parameterTypes.length; j++) {
                        System.out.print(parameterTypes[j].getName());

                        // If this isn't the last parameter, append ','
                        If (j < parameterTypes.length-1) {
                                System.out.print(",");
                        }
                }
                System.out.println(");");
        }
        System.out.println("}");
}
```

[0027] In an exemplary embodiment of the present invention, executing the MAIN function, including invoking the printInterface method for man.interface ("java.lang.Interface" class instance of the Human interface meta-data class) and bird.interface ("java.lang.Interface" class instance of the Bird interface meta-data class) will result in the following output:

```
interface Human {
        Destination walk(Destination target);
        Human talk(Human target);
        WriteableObject write(WriteableObject target);
}
interface Bird {
        Destination fly(Destination target);
        Bird talk(Bird target);
}
```

This output will be created even if the Destination and WriteableObject types are not available during runtime.

[0028] FIG. 4 is an exemplary process flow for providing a meta-data programming language level interface. At step 402, a "java.lang.Interface" Java interface meta-data class (either via interface name or a running Java object using the "interface" keyword hook) is received from a client program via a meta-data retrieval API. At step 404, the meta-data associated with the interface is requested. As described above in reference to Java, this information may be located in a Java byte array based on data located by a ClassLoader instance and entered into the Java byte array. Different programming languages will store and locate the interface (or abstract class) source data in different manners. The meta-data class will call a method associated with the correct programming language to retrieve the correct data. At step 406, the meta-data associated with each method contained in the interface is received. At step 408, the interface name is output. Next, at step 410, meta-data, such as method name, return type and parameters types are output.

[0029] In an alternate exemplary embodiment another hook may be utilized to provide an interface name to retrieve a Java interface meta-data class. Example code for directly using an interface name includes:

```
java.lang.Interface bird = java.lang.Interface.forName("Bird");
// special built-in static method "forName" returns an instance of "java.lang.Interface"
for Bird interface meta-data class
```

Example code for directly using a Java object via an "interface" keyword hook includes:

```
Human man = (Human) clark;
Bird bird = (Bird) clark;
printInterface(man.interface);
// where man.interface is an instance of "java.lang.Interface" for the Human interface
meta-data class
printInterface(bird.interface)
// where bird.interface is an instance of "java.lang.Interface" for the Bird interface meta-
data class
```

[0030] An exemplary embodiment of the present invention may be used by WebSphere for zOS v.4.01 SL4 System Management for Java Programming Language to introspect the Enterprise Java Bean (EJB) method level deployment descriptors. In addition, the exemplary embodiment may be utilized to properly assign method level deployment descriptors without requiring an Enterprise Archive (EAR file which contains one or more EJBs) to contain all Java classes that each of the EJBs reference. In this manner, the zOS's system management deployment strategy where deployment of each EAR may be done separately from the deployment target application server (where each application being deployed is required to provide all referenced classes) may be supported.

[0031] Embodiments of the present invention may be utilized to provide dynamic information about what an object is capable of handling. Instead of being forced through a database to collect meta-data (e.g., CORBA implementation) embodiments of the present invention allow the meta-data information to be accessed via a program during runtime. In this manner, the client may dynamically determine the capabilities of a particular object's interface.

[0032] As described above, the embodiments of the invention may be embodied in the form of computer-implemented processes and apparatuses for practicing those processes. Embodiments of the invention may also be embodied in the form of computer program code containing instructions embodied in tangible media, such as floppy diskettes, CD-ROMs, hard drives, or any other computer-readable storage medium, wherein, when the computer program code is loaded into and executed by a computer, the computer becomes an apparatus for practicing the invention. An embodiment of the present invention can also be embodied in the form of computer program code, for example, whether stored in a storage medium, loaded into and/or executed by a computer, or transmitted over some transmission medium, such as over electrical wiring or cabling, through fiber optics, or via electromagnetic radiation, wherein, when the computer program code is loaded into and executed by a computer, the computer becomes an apparatus for practicing the invention.

[0033] While the invention has been described with reference to exemplary embodiments, it will be understood by those skilled in the art that various changes may be made and equivalents may be substituted for elements thereof without departing from the scope of the invention. In addition, many modifications may be made to adapt a particular situation or material to the teachings of the invention without departing from the essential scope thereof. Therefore, it is intended that the invention not be limited to the particular embodiment disclosed as the best mode contemplated for carrying out this invention, but that the invention will include

all embodiments falling within the scope of the appended claims. Moreover, the use of the terms first, second, etc. do not denote any order or importance, but rather the terms first, second, etc. are used to distinguish one element from another.